Implementing and Benchmarking a Parallel LDL^T Factorization with SYCL

Michael Paciullo, Geraud Krawezik
Scientific Computing Core, Flatiron Institute

FLATIRON INSTITUTE
Scientific Computing Core

Rationale

Numerical linear algebra has applications in the fields of machine learning, geometric processing, and physical simulations. To solve a system of linear equations numerically, factorizing a matrix with methods such as LDU and QR is preferred for accuracy and speed. Once factorized into LDU or LDL^T, solving a system of linear equations only takes $O(n^2)$ additional floating point operations (FLOPs) for forward and backward substitution. For symmetric matrices, the LDL^T decomposition may be preferred over the Cholesky, or LL^T, decomposition because it requires no square root operations, and therefore works for matrices that have negative eigenvalues.
Most large systems of linear equations use sparse matrix storage and matrix algorithms. These sparse algorithms are based on, and are often implemented with, underlying dense matrix algorithms. GPUs are optimized for dense linear algebra[1], especially matrix-matrix operations. The goal of this study is to measure the performance of the SYCL language, a cross-platform GPU programming language that is a superset of C++, against the platform-specific implementations provided by NVIDIA's cuSolverDN. Here, we focus on the LDL^T decomposition [2] for a symmetric matrix A with no non-singular principal minors, or "upper-left square corners".

Algorithms used for factorization

If $A \in \mathbb{R}^{n\times n}$ is a symmetric matrix, and all principal minors of A are non-singular, then A can be factorized into LDL^T where L is a lower-triangular matrix with all diagonal elements equal to 1, and D is a diagonal matrix.

$$A = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ l_{31} & l_{32} & 1 & \\ \vdots & \cdots & & \ddots \end{bmatrix} \begin{bmatrix} d_1 & & & \\ & d_2 & & \\ & & d_3 & \\ & & & \ddots \end{bmatrix} \begin{bmatrix} 1 & l_{21} & l_{31} & \cdots \\ & 1 & l_{32} & \vdots \\ & & 1 & \\ & & & \ddots \end{bmatrix}$$

$$= \begin{bmatrix} d_1 & & & \\ l_{21}d_1 & d_2 & & \\ l_{31}d_1 & l_{32}d_2 & d_3 & \\ \vdots & \cdots & & \ddots \end{bmatrix} L^T = \begin{bmatrix} d_1 & l_{21}d_1 & l_{31}d_1 & \\ l_{21}d_1 & l_{21}^2 d_1 + d_2 & l_{21}l_{31}d_1 + l_{32}d_2 & \\ l_{31}d_1 & l_{21}l_{31}d_1 + l_{32}d_2 & l_{31}^2 d_1 + l_{32}^2 d_2 + d_3 & \\ \vdots & \cdots & & \ddots \end{bmatrix}$$

In general, for $a_{ij}, i, j \in \mathbb{N}, i > j$,

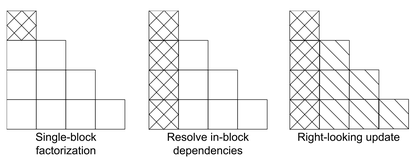
$$a_{ij} = \sum_{k=1}^{j} l_{ik}l_{jk}d_k = \sum_{k=1}^{j-1} l_{ik}l_{jk}d_k + l_{ij}d_j, \quad d_i = a_{ii} - \sum_{j=1}^{i-1} l_{ij}^2 d_j, \quad l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk}d_k}{d_j}$$

For example, $a_{43} = l_{41}l_{31}d_1 + l_{42}l_{32}d_2 + l_{43}d_3$. Similarly, $A \in \mathbb{R}^{mn\times mn}$ can also be factorized into block matrices.

$$A = \begin{bmatrix} ldlt^{-1}(D_1) & \cdots & \\ resolve^{-1}(L_{21})d_1 & L_{21}d_1L_{21}^T + ldlt^{-1}(D_2) & \cdots \\ resolve^{-1}(L_{31})d_1 & L_{31}d_1L_{21}^T + resolve^{-1}(L_{32})d_2 & L_{31}d_1L_{31}^T + L_{32}d_2L_{32}^T + ldlt^{-1}(D_3) \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$D_i = ldlt(A_{ii} - \sum_{j=1}^{i-1} L_{ij}D_jL_{ij}^T) \quad L_{ij} = resolve((A_{ij} - \sum_{k=1}^{j-1} L_{ik}d_kL_{jk}^T)d_j^{-1})$$

where m is the number of elements in a block, L_{ij} is an off-diagonal block of the result, D_i is a diagonal block of the result, d_i is D_i with only the diagonal elements included, *ldlt*() is the LDL^T factorization for a matrix, and *resolve*() is a subroutine that resolves all dependencies to the right for each column within a block. This algorithm requires a workspace of $m^2\frac{n(n+1)}{2}$ elements, and an auxiliary space of m^2n elements to store intermediary results of the dependency resolution step. Block indices are found using the triangular root of each column.



Right-looking updates

A non-diagonal block of A may look like

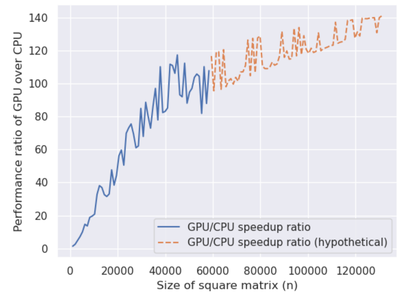
$$\begin{bmatrix} l_{51}d_1 & l_{51}d_1l_{21} + l_{52}d_2 & \cdots \\ l_{61}d_1 & l_{61}d_1l_{21} + l_{62}d_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}.$$

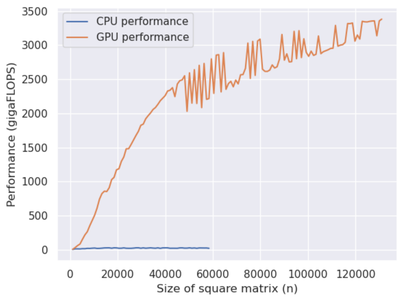
In order to extract the unique l values, for each row, and for each element starting from the left, the element in column i is stored in an auxiliary matrix, and divided by d_i in the workspace matrix. Then, all the elements to the right are subtracted from by the $l_{ji}d_i$ element stored in the auxiliary matrix multiplied by another l element from the diagonal block at the top of the current column. We called this subroutine *resolve*().
The final step for each column is to multiply the Ld blocks stored in the auxiliary space by the newly-derived L^T blocks in the main workspace. These are dense matrix-matrix multiplications, which are easily parallelizable and highly optimized on GPUs. This updates all blocks to the right of the current column. To help with cache locality, blocks from the left will no longer be read from or modified.

Comparison to NVIDIA

The performance of our software was measured against NVIDIA's implementation of a similar routine, sytrf(), which does not use packed storage, as part of their implementation of LAPACK, cuSolverDN. The timing for both versions includes allocating workspace on the GPU and copying the matrix to and from the GPU. Aside from small matrices, NVIDIA's software is up to 1.8× faster. However, our SYCL implementation is more memory-efficient due to the use of packed storage, using half as much memory on the CPU and GPU, in addition to being cross-platform.

Numerical results





The parallel block LDL^T algorithm using a packed data structure, called ssptrf() in LAPACK, was implemented in both OpenMP, a CPU parallelization library running on an Intel Xeon with 32 threads allocated, and SYCL, a cross-platform open standard GPU library, running on an NVIDIA H100 GPU. Even for small matrices, the GPU version vastly outperforms the CPU version. By the time the matrix is of size $40k \times 40k$, a 100-fold speedup by the SYCL implementation over OpenMP is observed, achieving speeds of more than 3 TeraFLOPS (floating point operations per second). Starting at $mn = 49152$, "jaggedness" in the performance graph is observed. This may be caused by the GPU cores being fully occupied. Possible future directions include adding a pivoting routine to the SYCL version for increased numerical stability, and implementing more packed symmetric LAPACK routines in SYCL for cross-platform support.

References

[1] Vasily Volkov and James W. Demmel. "Benchmarking GPUs to tune dense linear algebra". In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 2008, pp. 1–11. DOI: 10.1109/SC.2008.5214359.

[2] Geraud Krawezik and Gene Poole. "Accelerating the ANSYS Direct Sparse Solver with GPUs". In: June 2009.